Recursion Theorem;
Other Models of Computation

CS 81: Computability and Logic
November 30, 2010
The Recursion Theorem
Recursion Theorem
(Kleene 1938)

\textbf{Informally}: A program can have access to its own description (code).

\begin{itemize}
  \item Take input $x$.
  \item \ldots
  \item put $\langle S \rangle$ on the tape
  \item \ldots
  \item do stuff with $x$ and $\langle S \rangle$
  \item \ldots
\end{itemize}
Recursion Theorem  
(Kleene 1938)

Formally:
If R is a Turing machine computing a binary function R(x, y), then there is a Turing machine S computing a unary function such that:

\[ S(x) = R(x, \langle S \rangle) \]

where \( \langle S \rangle \) is the description of S itself.
Application: Undecidability of $A_{TM}$ (again)

- Suppose $A_{TM}$ were decidable using TM A.

- Define $M(x)$ as follows:
  1. Take input $x$;
  2. Use A to decide whether $M$ accepts $x$, i.e., whether $\langle M, x \rangle \in A_{TM}$
  3. Return the opposite answer.

- $M$ can’t exist, so A must not exist. QED
Recursion Theorem to the rescue

Code for TM R

Take input x and y. Put y and x on tape. Run A as subroutine. Return opposite answer.

Code for TM M

Take input x. Put \langle M \rangle and x on tape. Run A as subroutine. Return opposite answer.
Even if a machine is not given a handle to its own code on its tape at the outset, there are ways for it to construct it.

Such programs are now called “Quines”
A Java Quine

http://www.knet.ro/lsantha/

```java
class Q{
    public static void main(String[] v) {
        char c = 34;
        System.out.print(s + c + s + c + ';' + '');//
    }
}
```
char f[] =
"char f[] = %c%c%s%c;%cmain() {printf(f,10,34,f,34,10,10);}%c"
main() {printf(f,10,34,f,34,10,10);}
rex Quine
(by a Pomona College Student)

a = "\"; aa = "a";
b = "\"; bb = "b";
c = "\"; cc = "c";
d = "print(
    aa, c, b, a,a,b, f,
    aa,aa, c,b,aa,b,f,g,bb,c,b,
    a,b,b,f bb,bb,c,b,bb,b,f,g,
    cc,c,b,c b,f, cc,cc,c,b,cc,
    b, f,g ,dd ,c,b,
    d, b,f , g,g,
    dd, dd, c,b,
    dd, b,f , g,ee
    ,c,b , e, b,f,
    ee, ee, c,b,
    ee,b,f , g,ff,c, b,f,
    b,f,ff,ff,c,b,ff,b,f, g,gg,
    c,b,a ,nn,b,
    f,gg gg,c,b,
    gg,b,f , g,nn,nn,c
    ,b,nn,b, f,g,g,d,g);"

print(
    aa, c, b, a,a,b, f,
    aa,aa, c,b,aa,b,f,g,bb,c,b,
    a,b,b,f bb,bb,c,b,bb,b,f,g,
    cc,c,b,c b,f, cc,cc,c,b,cc,
    b, f,g ,dd ,c,b,
    d, b,f , g,g,
    dd, dd, c,b,
    dd, b,f , g,ee
    ,c,b , e, b,f,
    ee, ee, c,b,
    ee,b,f , g,ff,c, b,f,
    b,f,ff,ff,c,b,ff,b,f, g,gg,
    c,b,a ,nn,b,
    f,gg gg,c,b,
    gg,b,f , g,nn,nn,c
    ,b,nn,b, f,g,g,d,g);
Applications of Quines

✓ Entertainment
✓ Computer viruses
✓ Artificial life?
Recursion from the Recursion Theorem

If you have $\langle M \rangle$ on the tape, you can run it.

A machine $M$ can put $\langle M \rangle$ on the tape.

Therefore, a TM can compute “recursively” in the modern sense.

---

**Code for TM R**

Take input $x$ and $M$. If $x = 0$, write 1 and halt.  
Otherwise, run $M$ on $(x-1)$ and $M$. Multiply result by $x$.

**Code for TM F**

Take input $x$. If $x = 0$, write 1 and halt. 
Otherwise, run $F$ on $(x-1)$. Multiply result by $x$. 

---

Recursion Theorem
Theorem (Sipser)

The language $\text{MIN} = \{ \langle M \rangle \mid M \text{ is minimal} \}$ is not even recognizable.

Minimal

$=$

no machine with the same behavior has smaller description.
Proof

- Suppose MIN were recognizable, hence enumerated by some machine E.

- Consider the TM C:
  - Take input x.
  - Start running enumerator E.
  - Stop when we produce a $\langle D \rangle$ that is strictly longer than $\langle C \rangle$ (This must happen. Why?)
  - Simulate D running on x.
- C cannot exist (why?), so E cannot exist.
- Thus MIN cannot be recognizable.
Other Models of Computation
Two-Stack Machine

- PDA with 2 stacks
- Transition can push or pop one stack
- Can transition based on the top symbol of one stack.
Theorem:
A Two-Stack Machine can simulate a TM

Proof idea:

✓ one stack to hold the symbols to the left of the head
✓ one stack to hold the symbols below and to the right.
✓ Moving the head corresponds to copying a symbol from one stack to the other.
Counter Machines (CM)

- A counter machine is like a TM or PDA, but instead of having a stack or tape, it has one or more integer counters (unbounded capacity)
  - Transitions can increment or decrement one counter
  - Transitions can test whether a counter is zero/nonzero

- Equivalent: PDA with one or more stacks, each having only one symbol (except $ at the bottom)
  - All that matters is the depth of each stack
  - Can only see the top of stack, so can only test empty/nonempty

- NB: Any language accepted by a CM is recognizable.
Theorem: A 3-counter machine can simulate a TM.

**Idea:** A 3-counter machine can simulate a 2-stack machine.

- Suppose our 2-stack machine uses $r$ different stack symbols. Wlog, call these symbols 0 .. $r-1$.
- Represent the stack $X_1X_2 \cdots X_n$ (with $X_1$ on top) by the number $X_nr^{n-1} + X_{n-1}r^{n-2} + \cdots + X_2r + X_1$
- Pop = divide by $r$; top symbol is the remainder.
- Push k = multiply by $r$ and add k.
- Two of the counters hold two (encoded) stacks.
- Arithmetic operations use the third counter as scratch space.
Theorem:
A 2-counter machine can simulate a TM.

Idea: A 2-counter machine can simulate a 3-counter machine.

Represent the 3 counters i, j, and k by the single number

$$2^i3^j5^k$$

All necessary arithmetic can be done using the second counter as
scratch space.
Partial Recursive Functions

✔ Purely mathematical formulations of functions of natural numbers.
  ✔ Small set: Primitive Recursive functions (total)
  ✔ Larger set: Partial Recursive functions (possibly partial)

✔ Theorem:
  ✔ A TM can compute exactly the partial recursive functions (partial because TM might not terminate on all inputs)
  ✔ Partial Recursive functions can (by encoding tapes as integers) simulate any TM.
**Primitive** Recursive Functions
(an inductive definition!)

- The constantly-zero functions
  (take \( k \geq 0 \) arguments; return 0)

- The projection functions
  (take \( k \geq 0 \) arguments; return the \( i^{th} \))

- The successor function
  (take one argument \( n \); return \( n+1 \))
The composition of prim. rec. functions is prim rec.
e.g., \( h(x,y) = f(g_1(x,y), g_2(x,y), g_3(x,y)) \)

Corollary: the constantly-7 functions is PRF, for any \( n \), because they are compositions of the constantly-zero function and successors.

Note: we often write explicit definitions like
\[ h(x,y) = f(g_1(y), g_2(x), 2) \]
rather than
\[ h(x,y) = f(g_1(\text{proj}_2(x,y)), g_2(\text{proj}_1(x,y)), \text{succ}(\text{succ}(\text{zero()}))) \]
If $b$ and $r$ are prim rec. (with appropriate # of arguments) then so is the function $f$ defined by

\[
\begin{align*}
f(0, x_1, ..., x_n) &= b(x_1, ..., x_n) \\
f(n+1, x_1, ..., x_n) &= r(x_1, ..., x_n, n, f(n, x_1, ..., x_n))
\end{align*}
\]

Note: this "primitive recursion" is a very stylized (restricted) template for coding recursively.
Examples of Primitive Recursive Functions

- plus(x, y)
  - plus(0, y) = y
  - plus(n+1, y) = succ(plus(n, y))

- times(x, y)
  - mult(0, y) = 0
  - mult(n+1, y) = plus(y, times(n, y))

- pred(x)

- monus(x, y)

- mod(x, y)

- div(x, y)

- sqrt(x)
  - fact(0) = 1
  - fact(n) = mult(n, fact(n-1))

- ?: Predicates like even, equality, etc. (return 0/1)
Primitive Recursion

✓ **Theorem:** Primitive Recursive functions are total.
✓ **Proof:** By induction.

✓ Are all total functions TM-computable?
✓ Are all TM-computable total functions primitive recursive?
Computable but not Primitive Recursive: Diagonalization

We (hence a TM) can enumerate all the primitive recursive functions $p_1, p_2, p_3, \ldots$

Then $q(n) = p_n(n) + 1$ is total and computable.
Computable but not Primitive Recursive: Ackermann Hierarchy

\[
\begin{align*}
A_0(m) &= S(m) \\
A_{n+1}(0) &= A_n(1) \\
A_{n+1}(m+1) &= A_n(A_{n+1}(m)) \\
A(m) &= A_m(m)
\end{align*}
\]

✓ **Theorem:**
A(m) is a total function that grows faster than any primitive recursive function of one argument.

\[
\begin{align*}
A(1) &= 3 \\
A(2) &= 7 \\
A(3) &= 61 \\
A(4) &= 2^{2^{65536}} - 3
\end{align*}
\]
Primitive Recursive vs. Partial Recursive Functions

(not to scale)
Partial Recursive Functions

✓ Prim. Rec. + a partial “minimization operator”

If $h(x_1, \ldots, x_n)$ is partial recursive, then

$$\mu k \ [h(k, x_2, \ldots, x_n) = 0]$$

is the function that (given $x_2, \ldots, x_n$) computes the least $k$ (if any) that makes $h(k, x_2, \ldots, x_n)$ zero.
Examples

\[ \text{sqrt}(n) = \mu k. [\text{monus}(n, \text{times}(k, k)) = 0] \]

\[ \text{diverge}(n) = \mu k. [\text{monus}(k+1, k) = 0] \]

\[ \text{strange}(m, n) = \mu k. [\text{not}(\text{eq}(k+m, n)) = 0] \]
Partial Recursion vs. TMs

Every Partial Recursive operation (including minimization) could be programmed in a TM.
We can encode TM configurations as numbers.

Primitive Recursive functions exist
- \( R(x) = \) configuration one step beyond configuration \( x \).
- \( T(i,x) = \) configuration \( i \) steps beyond configuration \( x \).
- \( P(x) = \) whether configuration \( x \) is halting (0 or 1)

Computation length = \( \mu i. [P(T(i, x_0)) = 0] \)

Final configuration = \( T(\mu i. [P(T(i, x_0)) = 0], x_0) \)
Enumerating Partial Recursive Functions

For each $k$, the Partial Recursive Functions with $k$ arguments can be enumerated $f_1, f_2, \ldots$

There is even a $k+1$-argument PRF that given $i$, computes $f_i$ applied to the $k$ remaining arguments.

Why doesn’t the diagonal argument work again, giving us a computable function not in this list?

{ $j \mid f_j(\bullet)$ is total } is not recognizable/enumnerable.

{ $j \mid f_j(j)$ is defined } is recognizable, not decidable.